SIMULATOR FOR SOFTWARE DEVELOPMENT AND RECORDING MEDIUM HAVING SIMULATION PROGRAM RECORDED THEREIN

BACKGROUND OF THE INVENTION

[0001] 1. Field of the Invention

[0002] The present invention relates to a simulator designed for development of target processor-adapted built-in software, and an art related thereto.

[0003] 2. Description of the Related Art

[0004] In general, a software development system run on a host processor is used to develop target processor-adapted built-in software. The host processor is incompatible with a target processor.

[0005] The term "host processor" as set forth herein actuates the software development system. The host processor is used for software development and the verification of the resulting software.

[0006] The term "target processor" as given herein differs from the host processor. The target processor is used to execute development results or rather the resulting software.

[0007] The host processor is incompatible in software with the target processor. The resulting software normally runs only on the target processor, not on the host processor.

[0008] The term "simulator" as set forth herein creates resulting software-like, simulated software that is executed on the host processor, not on the target processor.

[0009] As described above, the software designed for the target processor that differs from the host processor is developed using the host processor-adapted software development system. However, such software development involves problems as discussed below.

[0010] A recent trend of the above-discussed software development has been changed, with an increase in scale of software, from assembler language-driven development to high-level language-driven development (e.g., C/C++).

[0011] This is because the software development using high-level languages allows processing such as data retention, transfer, and computation to be described independently of target processor assembler instructions and resources such as a register and a memory. This means that the high-level language-led software development is superior in readability, versatility, and development efficiency.

[0012] In particular, built-in software must be optimized to maximize processor capability in order to provide utmost system performance.

[0013] However, the high-level language-based software development has a compiler performance problem that generates a redundant code upon transformation from any high-level language into an assembler code. This problem may adversely affect software code size and execution speed.

[0014] In order to smooth out the problem, software development associated with heavy load-involving processing such as signal processing is still now made using a target processor-adapted assembler language in addition to the high-level language-based software development.

[0015] The use of the target processor-adapted assembler language makes the efficiency of software development low.

[0016] In a field of build-in software development, since the software development is large in scale and complicated, production process tends to be huge. In addition, increase of verification process and simulation time is also remarkable.

[0017] In many cases, the software development must be started, being unable to obtain the target processor, since the target processor has not completed yet.

[0018] Performance of the software should be analyzed also in a phase that software development environment is not prepared enough.

[0019] However, it is difficult to get precise information for a short time, since a simulator cannot run rapidly enough to meet with large-scaling and complication of LSIs.

[0020] According to the prior art, preparing software development environment (e.g. a compiler and a simulator) needs a lot of man-days. It tends to be delayed to start development of target software using the software development environment.

OBJECTS AND SUMMARY OF THE INVENTION

[0021] In the view of above, an object of the present invention is to provide simulators (compiler type and interpreter type) that can help software development more efficiently than the prior art.

[0022] More particularly, the present invention provides simulators that can simulate, optimizing both code size and executing-speed in an assembler level, using common software described all in a high-level language.

[0023] A first aspect of the present invention provides a simulator comprising: a compiler operable to compile a source code described in a high-level language; and a library including functions and/or procedures that are defined in the high-level language and model components of a target processor different from a host processor, wherein the source code is described using the library.

[0024] In the library, the hardware-components of the target processor are modeled using the functions and/or the procedures. The source code is described using the library. Thereby, a compiler type simulator can be constituted.

[0025] Since an object-code that the compiler type simulator outputs can be executed without passing through processes, such as instruction-fetching processes and instruction-decoding processes, executing-speed is high.

[0026] Since all source code is described in the high-level language, the development is excellent in portability or readability and the software development, including optimization in an assembler level, can be done all in the high-level language.

[0027] As mentioned later, this source code can be used in both the compiler type simulator and the interpreter type simulator. In development of two types (compiler type and interpreter type) of simulators, common parts can be used, thereby improving

development efficiency.

[0028] A second aspect of the present invention provides a simulator as defined in the first aspect of the present invention, wherein the components of the target processor further comprising: an accumulator of the target processor; a memory controller of the target processor; and a register of the target processor.

[0029] With this structure, the functions and/or the procedures can express behavior of an accumulator and a memory controller that are important components of the target processor.

[0030] A third aspect of the present invention provides a simulator as defined in the first aspect of the present invention, wherein the library further comprising: a hardware model library defining in the high-level language the functions and/or the procedures that model the components of the target processor; and an instruction-set-library defining in the high-level language functions and/or procedures corresponding to instructions of the target processor using the functions and/or the procedures of the hardware model library.

[0031] In this structure, since the library is divided into the hardware model library and the instruction-set-library, thereby handling of the library becomes easy.

[0032] That is, the hardware model library is shared with an interpreter type simulator mentioned later, and the instruction-set-library is added to the compiler type simulator.

[0033] In other words, the compiler type simulator according to this structure is constructed only by adding the instruction-set-library to the hardware model library, and since the translator and so on are unnecessary, the compiler type simulator can be more simply constituted than the interpreter type simulator.

[0034] For this reason, man-days of simulator development decrease and the simulator can be supplied to a developer early.

[0035] A fourth aspect of the present invention provides a simulator as defined in the third aspect of the present invention, wherein the instructions of the target processor

comprises an ADD instruction, an SUB instruction, an AND instruction, an OR instruction, an LD instruction, an ST instruction, an SET instruction and an MOV instruction.

[0036] With this structure, important instructions of the target processor can be covered.

[0037] A fifth aspect of the present invention provides a simulator as defined in the first aspect of the present invention, wherein the functions and/or the procedures of the library further comprising a function and/or a procedure calculating at least one of an executing-cycles-number of the target processor and power consumption of the target processor.

[0038] With this structure, the functions and/or procedures that are included in the library calculate and output an executing-cycles-number, power consumption, and so on. When simulation terminates, the executing-cycles-number of an assembler program, power consumption, and so on can be obtained, and performance can be analyzed in the compiler type simulator.

[0039] A sixth aspect of the present invention provides a simulator as defined in the first aspect of the present invention, wherein at least one of an executing-cycles-number of the target processor and power consumption of the target processor can be changed.

[0040] With this structure, changing intentionally the executing-cycles-number, power consumption can perform the simulation under various conditions.

[0041] A seventh aspect of the present invention provides a simulator as defined in the first aspect of the present invention, the functions and/or the procedures of the library further comprising a function and/or a procedure calculating code size in the target processor.

[0042] With this structure, optimization in code size and performance analysis can be performed.

[0043] An eighth aspect of the present invention provides a simulator comprising: a translator operable to read an source code described in a high-level language to output

an object-code; an instruction-fetching unit operable to fetch the object-code to output an fetched object-code; an instruction-decoding unit operable to output an decoded object-code; an executing unit operable to execute the decoded object-code; and a library including functions and/or procedures that are defined in the high-level language and model components of a target processor different from a host processor, wherein the source code is described using the library.

[0044] With this structure, an interpreter type simulator can be constituted.

[0045] In addition, the source code of the interpreter type simulator can be used in the compiler type simulator mentioned above. Since programs being executed in two types of simulators (compiler type and interpreter type) can be unified to one common program, thereby development efficiency of the programs can be improved compared with the prior art.

[0046] The above, and other objects, features and advantages of the present invention will become apparent from the following description read in conjunction with the accompanying drawings, in which like reference numerals designate the same components.

BRIEF DESCRIPTION OF THE DRAWINGS

[0047] Fig. 1 is a block diagram of software development environment according to a first embodiment of the present invention;

[0048] Fig. 2 is a block diagram of the target processor according to the first embodiment of the present invention;

[0049] Fig. 3 (a) is an illustration showing an example of the instructions of the target processor according to the first embodiment of the present invention;

[0050] Fig. 3 (b) is an illustration showing an example of register assignment of the target processor according to the first embodiment of the present invention;

[0051] Fig. 4 (a) is a model drawing of an accumulator according to the first embodiment of the present invention;

[0052] Fig. 4 (b) is a model drawing of a memory controller according to the first embodiment of the present invention;

[0053] Fig. 5 is an illustration showing an example of a header file of the hardware model library according to the first embodiment of the present invention;

[0054] Fig. 6 is an illustration showing an example of an implement file of the hardware model library according to the first embodiment of the present invention;

[0055] Fig. 7 is an illustration showing an example of the implement file of the instruction-set-library according to the first embodiment of the present invention;

[0056] Fig. 8 (a) is an illustration showing an example of a header file of the instruction-set-library according to the first embodiment of the present invention;

[0057] Fig. 8 (b) is an illustration showing an example of the source code according to the first embodiment of the present invention;

[0058] Fig. 9 (a) is a flow chart of processing of the interpreter type simulator according to the first embodiment of the present invention;

[0059] Fig. 9 (b) is a flow chart of processing by an assembler;

[0060] Fig. 10 is an illustration showing an example of an implement file of the interpreter type simulator according to the first embodiment of the present invention;

[0061] Fig. 11 is an illustration showing an example of an implement file of the hardware model library according to the second embodiment of the present invention;

[0062] Fig. 12 is an illustration showing an example of an implement file of the instruction-set-library according to the second embodiment of the present invention;

[0063] Fig. 13 is an illustration showing an example of a source code according to the second embodiment of the present invention;

[0064] Fig. 14 is an illustration showing an example of an implement file of the instruction-set-library according to a third embodiment of the present invention;

[0065] Fig. 15 (a) is an illustration showing an example of arrays according to the third embodiment of the present invention; and

[0066] Fig. 15 (b) is an illustration showing an example of a source code according to the third embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0067] Preferred embodiments of the present invention are now described in conjunction with the accompanying drawings.

[0068] In advance of concrete explanation, shortly, a high-level language in this specification is explained and functions and/or procedures thereof are described. In each following embodiment, C-language is used as the high-level language.

[0069] Herein, on the specification of C-language, not only a process with a return value but also a process without a return value (void type) is called a "function".

[0070] However, in other high-level languages, for example, Pascal, a process with a return value is called a "function", but a process without a return value is called a "procedure", and procedures and functions are strictly distinguished.

[0071] Here, the high-level language in this specification is not limited to C-language, but includes one of other high-level languages well known, for example, C++ language having upward compatibility with C-language, Pascal, and so on.

[0072] Taking Pascal into consideration, in the library of this specification, hardware-components of the target processor are modeled by "functions and/or procedures", in general. Each of these functions and/or procedures corresponding to one of instructions of the target processor is defined and described using the high-level language.

[0073] (First Embodiment)

[0074] Next, the concrete configuration of this embodiment will now be explained.

[0075] Fig. 1 is a block diagram of the software development environment in the first embodiment of the present invention.

[0076] Roughly speaking, this development environment comprises two kinds of simulators of a compiler type simulator 102 and a interpreter type simulator 108.

[0077] Of course, a simulator of the present invention can also make a compiler type simulator or an interpreter type simulator independently. Such independently made simulators are also included in the present invention.

[0078] As shown in Fig. 1, both of the compiler type simulator 102 and the interpreter type simulator 108 use a same hardware model library 101.

[0079] The hardware model library 101 defines functions, variables, and so on, modeling the hardware-components (accumulators, memory controllers, registers, and so on.) of the target processor in C-language, as mentioned below in detail using an example of a source code.

[0080] An instruction-set-library 105 defines, in C-language, functions corresponding to instructions of the target processor, using functions in the hardware model library 101.

[0081] The library of this embodiment comprises the hardware model library 101 and the instruction-set-library 105.

[0082] For performance analysis of software using the compiler type simulator 102, functions that calculate an executing-cycles-number, power consumption, resource usage, and so on are included in the library.

[0083] The hardware model library 101 is applied common to two kinds of software simulators, the compiler type simulator 102 and the interpreter type simulator 108.

[0084] In the library, variables and functions realizing hardware components in the target processor are prepared. The variables and functions in the library are used as components of both the compiler type simulator and the interpreter type simulator.

[0085] Next, the compiler type simulator 102 is explained in general.

[0086] A source code 103 is a C-language program described using definitions of the hardware model library 101 and the instruction-set-library 105. This source code 103 is also a source code of the interpreter type simulator 108.

[0087] A compiler 104 is a compiler of C-language that runs on the host processor. The compiler 104 does not have to be specific, one of usual C-compilers may be used as this

compiler 104. The compiler 104 compiles the source code 103 to output an object file 106.

[0088] The instruction-set-library 105 has already been compiled and has become a form of an object file, before compiling the source code 103.

[0089] A linker 100 links the object file 106 and the instruction-set-library 105 (object file) to output an object-code 107 of a machine language of the host processor.

[0090] As mentioned above, this object-code 107 operates on the host processor, and, does not operate on the target processor in general.

[0091] Next, the interpreter type simulator 108 will now be explained.

[0092] A translator 112 translates the source code 103 described in C-language into an object-code 113 for the target processor.

[0093] When executing, the instruction-fetching unit 109 reads one instruction of the object-code 113 to output the fetched object-code correspond to the instruction, the instruction-decoding unit 110 decodes the fetched object-code to output decoded object-code correspond to the instruction, and the executing unit 111 executes the decoded object-code.

[0094] Referring now to Fig. 2, a configuration of the target processor assumed with this embodiment is explained. Fig. 2 is a block diagram of the target processor used by the first embodiment of the present invention.

[0095] Fig. 2 shows just an example of the target processor of course and the present invention can be similarly applied to the target processors that take other configurations.

[0096] The target processor shown in Fig. 2 comprises the following components.

[0097] An instruction register (IR) 201 is a register holding an executing instruction.

[0098] A program counter (PC) 205 is a register holding an address of the executing instruction.

[0099] Registers 206 hold data and operation results that have been read from a memory mentioned later, and are constituted by four 16-bits registers in this embodiment.

[0100] These registers 206 are called a register R0 (209), a register R1 (210), a register R2 (211), and a register R3 (213), respectively.

[0101] As shown in Fig. 4 (a), an accumulator (ALU) 207 is a unit that performs four kinds of 16-bits binary operations (addition (ADD), subtraction (SUB), logical operations (AND, OR), and comprises two data inputs (IN1, IN2), one data output (OUT), and one control signal (CTL).

[0102] As shown in Fig. 4 (b), a memory controller 208 is a unit that controls access to data memory A (203) and data memory B (204), and selects a data memory to be accessed using an address value.

[0103] In Fig. 4 (b), an AD signal expresses an address value of a memory to be accessed. A DB signal is a pointer.

[0104] When reading, the memory controller 208 reads data from a memory variable and stores the data in a pointer pointed out by the DB signal.

[0105] When writing, the memory controller 208 reads data pointed out by the DB signal and stores the data in the memory variable.

[0106] RW signal is a control signal showing read/write.

[0107] In Fig. 2, the following memories are connected to the target processor via a bus 200.

[0108] An instruction memory (IMEM) 202 is a memory for storing a program, and a data memory A (203) and a data memory B (204) are memories in which the target processor stores data for calculation.

[0109] Referring now to Fig. 3, instructions of the target processor will be explained. Fig. 3 (a) is an illustration showing an example of the instructions of the target processor in the first embodiment of the present invention.

[0110] As shown in Fig. 3 (a), this target processor has eight kinds of instructions, an ADD instruction, an SUB instruction, an AND instruction, an OR instruction, an LD instruction, an ST instruction, an SET instruction, and an MOV instruction.

[0111] The target processor follows rules shown in Fig. 3 (b), concerning assignment of registers.

[0112] In the ADD instruction, the target processor adds data stored in a second-operand register to data stored in a first-operand register to store a result in the first-operand register.

[0113] In the SUB instruction, the target processor subtracts data stored in the second-operand register from data stored in the first-operand register to store a result in the first-operand register.

[0114] In the AND instruction, the target processor calculates an AND of data stored in the first-operand register and data stored in the second-operand register to store a result in the first-operand register.

[0115] In the OR instruction, the target processor calculates an OR of data stored in the first-operand register and data stored in the second-operand register to store a result in the first-operand register.

[0116] In the LD instruction, regarding data stored in a register assigned by the second-operand register as an address of one of the data memory A (203) and the data memory B (204), the target processor reads data at the address and stores the data into the first-operand register.

[0117] In the ST instruction, the target processor regards data in the second-operand register as an address of one of the data memory A (203) and the data memory B (204), and stores the data in the first-operand register into the address.

[0118] In the SET instruction, the target processor stores an immediate data assigned by the second operand into the first-operand register.

[0119] In the MOV instruction, the target processor stores data in the second-operand register into the first-operand register.

[0120] Referring now to Fig. 5 and Fig. 6, an example of a configuration of the hardware model library 101 that models each component of the target processor shown

in Fig. 2 is described.

[0121] Fig. 5 is an illustration showing an example of a header file of a hardware model library, and Fig. 6 shows an example of an implement file of the hardware model library in the first embodiment of the present invention.

[0122] The same name is given to each of the program components corresponding to the components in Fig. 2 and Fig. 3.

[0123] For example, a name of "IMEM" is given to a program component corresponding to the instruction memory (IMEM) 202 in Fig. 2.

[0124] Declarations of variables and/or functions are made using the header file as shown in Fig. 5, and the implement file of Fig. 6 has been described concerning the contents of the variables and the functions.

[0125] However, it is not necessary to divide the hardware model library into the header file and the implement file, that is, all items of the hardware model library may be described in one file.

[0126] As shown in Figs. 4 and 5, arrays IMEM, DMEMA, and DMEMB express instruction memory 202, data memory A (203), and data memory B (204), each having components described by the respective sizes.

[0127] Of course, a data structure that models one of the memories 202, 203 and 204 does not have to be an array, and other well known structures (e.g. list) may be used instead.

[0128] The variables IR, PC, R0, R1, R2 and R3 express, respectively, an instruction register (IR) 201, a program counter (PC) 205, a register (R0) 209, a register (R1) 210, a register (R2) 211, and a register (R3) 213, and are defined as short type variables, that is, 16-bits integer type.

[0129] As shown in Fig. 4 (a), an ALU function having four arguments of variables IN1, IN2, OUT, and CTL expresses an accumulator (ALU) 207.

[0130] The variables IN1 and IN2 express two data inputs and are short type pointers.

[0131] The variable OUT expresses a data output and is a short type pointer.

[0132] The variable CTL expresses a control signal and is an int type variable.

[0133] The ALU function is described in C-language such that the ALU function has the same calculation precision as the target processor.

[0134] As shown in Fig. 4 (b), an MEMC function having three arguments of variables AD, DB, and RW expresses the memory controller (MEMC) 208.

[0135] The variable AD expresses an address value to be accessed.

[0136] The variable DB is a pointer. The MEMC function, when reading, reads data from a memory variable to store the data in a pointer assigned by the variable DB, when writing, the MEMC function reads data assigned by the variable DB to store the data in the memory variable.

[0137] The variable RW is a control signal showing read/write.

[0138] In this embodiment, the hardware model library 101 comprises the above nine variables (arrays are also included) IMEM, DMEMA, and DMEMB, IR, PC, R0, R1, R2 and R3, and two library functions (ALU function, MEMC function). Needless to say, more or less variables and functions may be used.

[0139] Referring now to Fig. 7, an example of an implement of the instruction-set-library 105 is explained, not in an object-code level but in a source code level.

[0140] As mentioned above, and as shown in Fig. 3 (a), the target processor has eight kinds of instructions, that is, an ADD instruction, an SUB instruction, an AND instruction, an OR instruction, an LD instruction, an ST instruction, an SET instruction, and an MOV instruction.

[0141] In the instruction-set-library 105 of this embodiment, the following functions corresponding to each of these instructions are prepared.

[0142] As shown in Fig. 7, each of an ADD function, an SUB function, an AND function, and an OR function is defined as a function having two arguments RS1 and

RS2, and operation of these functions is carried out by calling an ALU function 410 that is a library function of the hardware model library 101.

[0143] Here, the argument RS1 corresponds to the first-operand register, and the argument RS2 corresponds to the second-operand register, respectively.

[0144] Each of the LD function and the ST function is defined as a function having two arguments RS1 and RS2, and operation of these functions is carried out by calling the MEMC function 415 that is a library function of the hardware model library 101 and controls memories.

[0145] The argument RS1 corresponds to the first-operand register, and the argument RS2 corresponds to the second-operand register, respectively.

[0146] The SET function is defined as a function having two arguments RD and IMD, and substitutes the value of the argument IMD for a variable *RS1 assigned by the argument RS1.

[0147] The MOV function is defined as a function having two arguments RD and RS, and substitutes the value of the argument RS2 for the variable *RS1 assigned by the argument RS1.

[0148] In this embodiment, the above eight instructions are prepared in the instruction-set-library 105.

[0149] The header file as shown in Fig. 8 (a) is prepared so that it may be easy to include the implement file of Fig. 7 in a source code.

[0150] Referring now to Fig. 8, an example of source code 103 shown in Fig. 2 is described.

[0151] In the source code 103 of Fig. 8, the header file shown in Fig. 8 (a) is included at the beginning, and hereinafter each of functions implemented (See Fig. 7) in the instruction-set-library 105 can be used.

[0152] The SET function, the MOV function are called in a main function, and the called functions perform predetermined processes.

[0153] In this source code 103, not only the functions contained in the instruction-set-library 105 but also the functions that can be defined in C-language can be described, of course.

[0154] With this structure, calls are performed in order of the source code 103, the instruction-set-library 105 and the hardware model library 101.

[0155] Simulation of operation of the target processor can be carried out by compiling and executing this source code 103 using a compiler 104 running on the host processor, in which such as a personal computer or a workstation is mounted.

[0156] Referring now to Fig. 9, explanation is added about the interpreter type simulator 108 shown in Fig. 1. First, a translator 112 is explained.

[0157] Fig. 9(a) shows a flow of processing by the interpreter type simulator 108 of this embodiment, and Fig. 9(b) shows a flow in a case of processing by an assembler 902, respectively.

[0158] As shown in Fig. 9 (a), the translator 112 of this embodiment needs to read the source code 103 (the same as the compiler type simulator 102) to output an object-code 113.

[0159] On the other hand, when there is an assembler program 901 corresponding to source code 103 (C-language), the assembler 902 assembles this assembler program 901 to output an object-code 903.

[0160] Herein, the purpose can be fulfilled if the object-code 113 is equal to the object-code 903.

[0161] Simply stated, it is sufficient if this translator 112 performs the following two processes (process 1 and process 2):

[0162] (process 1) The translator 112 inputs the source code 113 to replace the source code 103 with the assembler program 901; and

[0163] (process 2) The translator 112, having functions equivalent to the assembler 902, assembles the replaced assembler program 901.

[0164] More specifically, the process 1 can be performed by simple replacement of strings according to the following rules (rule 1, rule 2 and rule 3).

[0165] (rule 1) The translator 112 deletes lines including the string of "#include *", where "*" is a wild card.

[0166] (rule 2) The translator 112 replaces the string of "main () *" with the string of "main:".

[0167] (rule 3) The translator 112 replaces the string of "SET (&*, ?)" with the string of "SET *, ?", where "*" and "?" are wild cards.

[0168] In addition, the process 2 is easily performed substituting the assembler 902 for the functions equivalent thereto. Of course, specific functions equivalent to the assembler 902 may be provided for the translator 112.

[0169] The translator 112 needs not to perform the process 1 and the process 2 seperately, but may perform the processes 1 and 2 at once.

[0170] Referring now to Fig. 10, the instruction-fetching unit 109 of Fig. 1, the instruction-decoding unit 110 and the executing unit 111 will now be explained.

[0171] Fig. 10 is an illustration showing an example of the implement file of the interpreter type simulator in the first embodiment of the present invention. At the beginning of Fig. 10, the header file of the hardware model library 101 is included.

[0172] A type of variable "state" is declared enumerating only three values of "Fetch (=0)", "Decode", and "Exec".

[0173] Of course, the instruction-fetching unit 109 should operate when the variable "state" is "Fetch". Similarly, when the variables "state" is "Decode" or "Exec", the instruction-decoding unit 110 or the executing unit 111 should operate, respectively.

[0174] A variable "cycle" that memorizes an executing-cycles-number is declared as int type and introduced.

[0175] In a main function, after setting the variable "cycle" to "0" and setting the variable "state" to "Fetch", process starts.

[0176] In the next while sentence, an exec function is called and, at the end of the main function, a value of the variable "cycle" is outputted to the standard output. Thereby, the executing-cycles-number is acquired and measured.

[0177] Whenever the exec function is called, the variable "cycle" increases by one, hereinafter, in a switch sentence, process branches depending on the value of the variable "state".

[0178] When the variable "state" is "Fetch", process (from "case Fetch:" to just before "case Decode:") equivalent to the instruction-fetching unit 109 is performed.

[0179] When the variable "state" is "Decode", process (from "case Decode:" to just before "case Exec:") equivalent to the instruction-decoding unit 110 is performed.

[0180] When the variable "state" is "Exec", process (below "case Decode:") equivalent to the executing unit 111 is performed.

[0181] Note that variables, arrays, functions, and so on, such as the array IMEM and the ALU function, are defined in the hardware model library 101 and are used, in each of the above processes.

[0182] <Effects of First Embodiment>

[0183] As mentioned above, development using the compiler type simulator 102 and the interpreter type simulator 108 earns high efficiency than the prior art, since the simulators 102 and 108 comprises the library that models components of the target processor by the functions and/or the procedures.

[0184] The instruction-set-library 105 can be constructed, only preparing and compiling a common source code, easier than the interpreter type simulator 108. Thereby, man-days decrease, and the compiler type simulator 102 can be supplied earlier than the interpreter type simulator 108.

[0185] The translator 112 generates, from the source code 103 described by the functions of the instruction-set-library 105, the same object-code as an object-code that an assembler has assembled an assembler code. Using the translator 112, a common

source code can apply to both of two kinds of simulators 102 and 108.

[0186] (Second Embodiment)

[0187] Hereinafter, difference with a first embodiment is explained.

[0188] As shown in Fig. 11, in the hardware model library 101, a variable "cycle" showing an executing-cycles-number and a variable "power" showing power consumption are added.

[0189] In addition, in the ALU function, concerning ALU operation, an executing-cycles-number of ALU calculation is added to the variable "cycle" and power consumption of ALU calculation is added to the variable "power".

[0190] In the MEMC function, concerning memory access, an executing-cycles-number of the memory access is added to the variable "cycle" and power consumption of the memory access is added to the variable "power".

[0191] As shown in Fig. 12, in the instruction-set-library 105, in order to calculate the executing-cycles-number and the power consumption that are necessary for executing instructions using the SET function and the MOV function, a process adding the variables "cycle" and "power" is further included.

[0192] As shown in Fig. 13, in the source code 103, a sentence that carries out the standard output of the value of the variables "cycle" and "power" using a printf function is added.

[0193] Of course, instead of carrying out the standard output, the values of the variables "cycle" and "power" may be outputted to a file and so on. Thereby, information about the executing-cycles-number and the power consumption that program execution requires can be acquired and measured.

[0194] Although not illustrated, as for the interpreter type simulator 108 of the second embodiment, it is desirable to add, like the above, functions that carry out the standard output of the value of the variable "power" using a printf function, to the main function of Fig. 10.

[0195] Thereby, also in the interpreter type simulator 108, the information about the power consumption that program execution requires can be acquired and measured.

[0196] <Effects of Second Embodiment>

[0197] In addition to the effect of the first embodiment, since the library includes process that calculates the executing-cycles-number and the power consumption, making a corresponding variable output, the executing-cycles-number and the power consumption of the assembler program can be acquired and measured, when simulation ends. Performance can be analyzed in two kinds (compiler type and interpreter type) of simulators.

[0198] (Third Embodiment)

[0199] Only difference with the second embodiment is explained in a third embodiment. As shown in Fig. 14, in the instruction-set-library 105, a variable "cycle" showing an executing-cycles-number, a variable "power" showing power consumption, and a variable "code" showing code size are added.

[0200] In this embodiment, as shown in a table of Fig. 15(a), a unique index is given for every instructions, such as the ADD instruction and the SUB instruction, an increment of the variable "cycle" and an increment of the variable "power" are defined, and such increments are stored in an array cycle_tbl [] and an array power_tbl[], respectively. Of course, such increments may be stored in other storing construction that is not an array.

[0201] As shown in Fig. 14, in the instruction-set-library 105, process adding these variables "cycle", "power", and "code" is added in the ADD function, the SUB function, the LD function, the ST function, and the MOV function, respectively.

[0202] When the variables "cycle" and "power" are calculated, the values of arrays cycle_tbl [] and power_tbl[] that are necessary for each instruction execution, are used.

[0203] As shown in an example of source code of Fig. 15 (b), it is desirable to store data of values of arrays cycle_tbl [] and power_tbl[] in the file (in the example of illustration, "table" file), and to load, at the time of initialization, the values to arrays cycle_tbl []

and power_tbl[] from the file.

[0204] This init function is an initialization function to be executed before simulation starts.

[0205] <Effects of Third Embodiment>

[0206] In addition to the effect of the second embodiment, by adopting the mechanism of giving, from the outside (the file), information about the executing-cycles-number and the power consumption that instruction execution requires, it becomes able to change such information easily, and simulation according to various cases can be performed.

[0207] Although the example of description of the library that does not use object-orientation, the hardware model library 101 and the instruction-set-library 105 may be more smartly described with C++ language and so on, for example.

In all of the embodiments, the compiler type simulator 102 and the interpreter type simulator 108 can be supplied with "recording medium" (e.g. CD-ROM, FD, hard disk, and so on.) storing programs performing the compiler type simulator 102 and/or the interpreter type simulator 108.

[0208] Furthermore, the simulators 102 and 103 can be supplied with a personal computer, a workstation, and so on, to which the programs are pre-installed.

[0209] It is noted that the "recording medium" mentioned in this specification includes a case in which the programs are divided and stored in a plurality of sets of record media and distributed.

[0210] Additionally, regardless of whether or not the program is a part of an operating system, if the program causes some of the functions thereof to be performed by various processes or threads (DLL, OCX, Active X, and so on, (including the trademarks of Microsoft Corporation)), the "recording medium" includes a case in which a part concerning the functions performed by them is not stored in the recording medium.

[0211] A standalone type system is shown in Fig. 1. A server/client type system can be

used instead. In other words, instead of a case in which all components appearing in the specification are contained in only one terminal unit, a case is allowable in which one terminal unit is a client, and all of or a part of the components exist in a server or network connectable to the client.

[0212] A case is also allowable in which the server side has almost all the components of Fig. 1, and the client side has, for example, a WWW browser only. Normally, various kinds of information are located on the server, and are distributed to the client basically through a network. In this case, when necessary information is located on the server, a storage device of the server is the "recording medium" mentioned above, and, when the information is located on the client, a storage device of the client is the "recording medium".

[0213] In addition to an application that is compiled in a form of a machine language, the "digital signature program" includes a case in which it exists as an intermediate code interpreted by the aforementioned process or thread, a case in which at least a resource and a source code are stored on the "recording medium", and a compiler and a linker that can generate the application of a machine language from them are located on the "recording medium", or a case in which at least the resource and the source code are stored on the "recording medium", and an interpreter that can generate the application of the intermediate code from them is located on the " recording medium".

[0214] According to the present invention, in developing a compiler type simulator and an interpreter type simulator by modeling components of the target processor using the functions and/or the procedures, parts of simulators can be communalized and development efficiency improves.

[0215] Including optimization of assembler level, all of software development can be done in a high-level language.

[0216] The common source code can be used in both the compiler type simulator and the interpreter type simulator, by adding the translator mentioned above.

[0217] Also in the compiler type simulator, performance can be analyzed by including in the library a process that calculates the executing-cycles-number, the power consumption, and the code size. Such information can be changed easily and the simulation according to various cases can be done.

[0218] Having described preferred embodiments of the invention with reference to the accompanying drawings, it is to be understood that the invention is not limited to those precise embodiments, and that various changes and modifications may be effected therein by one skilled in the art without departing from the scope or spirit of the invention as defined in the appended claims.